Encapsulation

1. Student with Grade Validation & Configuration

Ensure marks are always valid and immutable once set.

- Create a Student class with private fields: name, rollNumber, and marks.
- Use a constructor to initialize all values and enforce marks to be between 0 and 100; invalid values reset to 0.
- Provide getter methods, but no setter for marks (immutable after object creation).
- Add displayDetails() to print all fields.

```
package Assesement_day5;

public class Student {
private String name;
private int rollno;
private int marks;
void details(String name,int rollno, int marks)
{
this.name=name;
this.rollno=rollno;
this.marks=marks;
System.out.println("Student Name=" +name);
System.out.println("Student Rollno="+rollno);
System.out.println("Student Marks=" +marks);
}

public static void main(String[] args) {
Student sd = new Student();
sd.details("sanjana",482,782);
```

```
}

}
```

Output:

```
Student Name=sanjana
Student Rollno=482
Student Marks=782
```

2. Rectangle Enforced Positive Dimensions

Encapsulate validation and provide derived calculations.

- Build a Rectangle class with private width and height.
- Constructor and setters should reject or correct non-positive values (e.g., use default or throw an exception).
- Provide getArea() and getPerimeter() methods.
- Include displayDetails() method.

```
package Assesement_day5;

public class Rectangle {

    private double width;

    private double height;

    public Rectangle(double width, double height) {

        this.width = (width > 0) ? width : 1;

        this.height = (height > 0) ? height : 1;

    }

    public double getArea() {

        return width * height;

    }
```

```
    public double getPerimeter() {
        return 2 * (width + height);
    }
    public void displayDetails() {
        System.out.println("Width: " + width);
        System.out.println("Height: " + height);
        System.out.println("Area: " + getArea());
        System.out.println("Perimeter: " + getPerimeter());
    }

    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5, 10);
        rectangle.displayDetails();
    }
}
```

Output:

Width: 5.0

Height: 10.0

Area: 50.0

Perimeter: 30.0

3. Inner Class Encapsulation: Secure Locker

Encapsulate helper logic inside the class.

- Implement a class Locker with private fields such as lockerId, isLocked, and passcode.
- Use an inner private class SecurityManager to handle passcode verification logic.
- Only expose public methods: lock(), unlock(String code), isLocked().
- Password attempts should not leak verification logic externally—only success/failure.
- Ensure no direct access to passcode or the inner SecurityManager from outside.public class Locker {

```
package Assesement_day5;
public class Locker {
	private boolean isLocked;
	private String passcode;
	public Locker(String passcode) {
		this.passcode = passcode;
		this.isLocked = true;
	}
	private class SecurityManager {
		boolean verifyPasscode(String code) {
			return passcode.equals(code);
		}
	}
	public boolean unlock(String code) {
```

```
        SecurityManager manager = new SecurityManager();
        if (manager.verifyPasscode(code)) {
            isLocked = false;
            return true;
        }
        return false;
    }
    public boolean isLocked() {
        return isLocked;
    }
    public static void main(String[] args) {
        Locker locker = new Locker("secret");
        System.out.println("Is Locked: " + locker.isLocked());
        System.out.println("Unlock: " + locker.unlock("wrong"));
System.out.println("Unlock: " + locker.unlock("secret"));
        System.out.println("Is Locked: " + locker.isLocked());
    }
}
```

Output:

```
Is Locked: true
Unlock: false
Unlock: true
Is Locked: false
```

5. Builder Pattern & Encapsulation: Immutable Product

Use Builder design to create immutable class with encapsulation.

- Create an immutable Product class with private final fields such as name, code, price, and optional category.
- Use a static nested Builder inside the Product class. Provide methods like withName(), withPrice(), etc., that apply validation (e.g. non-negative price).
- The outer class should have only getter methods, no setters.
- The builder returns a new Product instance only when all validations succeed.

```
package Assesement_day5;
    public class Product {
            private final String name;
            private final double price;
            private Product(Builder builder) {
                    this.name = builder.name;
                    this.price = builder.price;
            }
            public String getName() {
                    return name;
            }
            public double getPrice() {
                    return price;
            }
```

```
    public static class Builder {
        private String name;
        private double price;
        public Builder withName(String name) {
            this.name = name;
            return this;
        }
        public Builder withPrice(double price) {
            if (price < 0) {
                throw new IllegalArgumentException("Price
cannot be negative");
            }
            this.price = price;
            return this;
        }
        public Product build() {
            return new Product(this);
        }
    }
    public static void main(String[] args) {
        Product product = new Product.Builder()
                .withName("Apple")
                .withPrice(2.0)
                .build();
```

```
            System.out.println("Product: " + product.getName());
            System.out.println("Price: " + product.getPrice());
        }
    }
```

Output:

Product: Apple

Price: 2.0

Interface

1. Reverse CharSequence: Custom BackwardSequence

- Create a class BackwardSequence that implements java.lang.CharSequence.
- Internally store a String and implement all required methods: length(), charAt(), subSequence(), and toString().
- The sequence should be the reverse of the stored string (e.g., new BackwardSequence("hello") yields "olleh").
- Write a main() method to test each method.

```
package Assesement_day5;

public class BackwardSequence {
private String str;

public BackwardSequence(String str) {
this.str = new StringBuilder(str).reverse().toString();
}

public String getSequence() {
```

```
return str;
}

public static void main(String[] args) {
BackwardSequence seq = new BackwardSequence("My name is
sanjana");
System.out.println(seq.getSequence());
}

}
```

Output:

anajnas si eman yM

2. Contract Programming: Printer Switch

- Declare an interface Printer with method void print(String document).
- Implement two classes: LaserPrinter and InkjetPrinter, each providing unique behavior.
- In the client code, declare Printer p;, switch implementations at runtime, and test printing.

```
package Assesement_day5;
interface Printer {
void print(String document);
}
class LaserPrinter implements Printer {
public void print(String document) {
System.out.println("LaserPrinter: Printing " + document);
}
}
class InkjetPrinter implements Printer {

public void print(String document) {
System.out.println("InkjetPrinter: Printing " + document);
```

```
}
}
public class doc {

public static void main(String[] args) {
Printer p;

p = new LaserPrinter();
p.print("Doc 1");

p = new InkjetPrinter();
p.print("Doc 2");
}

}
```

Output:

```
LaserPrinter: Printing Doc 1
InkjetPrinter: Printing Doc 2
```

3. Extended Interface Hierarchy

- Define interface BaseVehicle with method void start().
- Define interface AdvancedVehicle that extends BaseVehicle, adding method void stop() and boolean refuel(int amount).
- Implement Car to satisfy both interfaces; include a constructor initializing fuel level.
- In Main, manipulate the object via both interface types.

```
package Assesement_day5;
interface BaseVehicle {
void start();
}

interface AdvancedVehicle extends BaseVehicle {
```

```
void stop();
boolean refuel(int amount);
}

class Car implements AdvancedVehicle {
private int fuelLevel;

public Car(int fuelLevel) {
this.fuelLevel = fuelLevel;
}

public void start() {
System.out.println("Car started");
}
public void stop() {
System.out.println("Car stopped");
}
public boolean refuel(int amount) {
fuelLevel += amount;
System.out.println("Car refueled. Current fuel level: " + fuelLevel);
return true;
}

public int getFuelLevel() {
return fuelLevel;
}
}

public class vehicle {

public static void main(String[] args) {
Car car = new Car(30);

BaseVehicle baseVehicle = car;
baseVehicle.start();
```

```
AdvancedVehicle advancedVehicle = car;
advancedVehicle.stop();
advancedVehicle.refuel(20);

System.out.println("Current fuel level: " + car.getFuelLevel());
}
}
```

Output:

```
Car started
Car stopped
Car refueled. Current fuel level: 50
Current fuel level: 50
```

6. Default and Static Methods in Interfaces

- Declare interface Polygon with:
    - double getArea()
    - default method default double getPerimeter(int... sides) that computes sum of side
    - a static helper static String shapeInfo() returning a description string
    - Implement classes Rectangle and Triangle, providing appropriate getArea().
    - In Main, call getPerimeter(...) and Polygon.shapeInfo().

```
package Assesement_day5;
interface Polygon {
double getArea();

default double getPerimeter(int...sides) {
double perimeter = 0;
for (int side : sides) {
```

```
perimeter += side;
}
return perimeter;
}
}
class Rectangle implements Polygon {
private double length;
private double width;

public Rectangle(double length, double width) {
this.length = length;
this.width = width;
}

public double getArea() {
return length * width;
}

public double getPerimeter() {
return Polygon.super.getPerimeter((int) length, (int) width, (int) length,
(int) width);
}
}
public class interfaces {

public static void main(String[] args) {
Rectangle rectangle = new Rectangle(4, 5);
System.out.println("Area: " + rectangle.getArea());
System.out.println("Perimeter: " + rectangle.getPerimeter());
}

}
```

Output:

```
Area: 20.0
Perimeter: 18.0
```

Lambda expressions

1. Sum of Two Integers

```
package Assesement_day5;
interface addition{
int addition(int a,int b);
}
public class sum_of_twodigits {

public static void main(String[] args) {
addition sum=(a, b) -> a +b;
System.out.println("Sum=" +sum.addition(8, 7));

}

}
```

Output:

Sum=15

2. Check If a String Is Empty

Create a lambda (via a functional interface like Predicate<String>) that returns true if a given string is empty.
Predicate<String> isEmpty = s -> s.isEmpty();

```
package Assesement_day5;

import java.util.function.Predicate;

public class String_empty {

public static void main(String[] args) {
Predicate<String> isEmpty = s -> s.isEmpty();
System.out.println(isEmpty.test(""));
System.out.println(isEmpty.test("Hello"));
```

```
}

}
```

Output:

true

false

3. Convert Strings to Uppercase/Lowercase

```
package Assesement_day5;

import java.util.function.Function;

public class strong_upper_lower {

public static void main(String[] args) {

Function<String, String> toUppercase = s -> s.toUpperCase();

Function<String, String> toLowercase = s -> s.toLowerCase();

String original = "This is my first project";

System.out.println("Original: " + original);

System.out.println("Uppercase: " + toUppercase.apply(original));

System.out.println("Lowercase: " + toLowercase.apply(original));

}

}
```

Output:

Original: This is my first project

Uppercase: THIS IS MY FIRST PROJECT

Lowercase: this is my first project

4. Sort Strings by Length or Alphabetically

```
package Assesement_day5;

import java.util.Arrays;

public class Sort_by_alphabetically {

public static void main(String[] args) {
String[] strings = {"monkey", "cat", "elephant", "dog", "lion"};

Arrays.sort(strings, (a, b) -> Integer.compare(a.length(), b.length()));
System.out.println("Sorted by length: " + Arrays.toString(strings));

Arrays.sort(strings, (a, b) -> a.compareTo(b));
System.out.println("Sorted alphabetically: " + Arrays.toString(strings));

}

}
```

Output:

```
Sorted by length: [cat, dog, lion, monkey, elephant]
Sorted alphabetically: [cat, dog, elephant, lion, monkey]
```

5.Create similar lambdas for max/min.

```
package Assesement_day5;

import java.util.function.BiFunction;

public class max_min {

public static void main(String[] args) {
BiFunction<Integer, Integer, Integer> max = (a, b) -> Math.max(a, b);
BiFunction<Integer, Integer, Integer> min = (a, b) -> Math.min(a, b);

System.out.println("Max: " + max.apply(10, 20));
```

```
System.out.println("Min: " + min.apply(10, 20));
}

}
```

Output:

Max: 20
Min: 10

6.Calculate Factorial

```
package Assesement_day5;

import java.util.stream.IntStream;

public class Factorial {

public static void main(String[] args) {
int n = 5;
int factorial = IntStream.rangeClosed(1, n)
.reduce(1, (a, b) -> a * b);
System.out.println(factorial);

}

}
```

Output:

120